

Time-Shifted Modules: Exploiting Code Modularity for Fine Grain Parallelization

Craig Zilles and Guri Sohi
Computer Sciences Dept., University of Wisconsin-Madison
1210 West Dayton St., Madison, WI 53706
[zilles, sohi]@cs.wisc.edu

Abstract

Multi-threaded processors and chip-multiprocessors execute concurrent threads in close physical proximity, potentially reducing the cost of synchronization and communication significantly and enabling the parallelization of programs at a fine grain. In this paper, we explore a source of fine-grain parallelism present in programs due to their modular nature. Concurrency is derived from executing code within a module in parallel with the main program. Because this technique exploits the modularity of code, rather than its regularity, it is applicable to irregular, integer applications. Furthermore, because all of the synchronization is encapsulated within the module, the process of parallelization is simplified—a programmer need only consider the module’s code—and, once created, libraries of such modules can be used to create parallel programs by programmers without having to reason about race conditions. We demonstrate the technique in two case studies, achieving speedups of 26% and 39% over the single-threaded base case on a simulated SMT processor.

1 Introduction

Due to the prevalence of thread-level parallelism (TLP) in server applications (e.g., databases, web servers), many next generation server processors will concurrently execute multiple threads in hardware [10, 11], using techniques like simultaneous multithreading (SMT) [30] and chip multi-processing (CMP) [24]. Historically, technology developed for high-end processors has been later deployed in commodity processors. Processor manufacturers often amortize design costs by reusing an existing processor core in a new market segment when process technology shrinks make it cost-effective to do so. Thus, we expect mainstream personal computer and workstation processors to support the concurrent execution of multiple threads in the coming years.

The problem is that most existing interactive workloads do not have TLP to exploit. Previous studies have shown that there are low levels of concurrency in most desktop applications [21] with

much of the existing concurrency coming from work delegated to the GUI process [15]. Although this lack of TLP in desktop applications can be partially attributed to the scarcity of multi-processor desktop machines to exploit it (a chicken and egg problem), the difficulty of manually parallelizing applications is probably the dominant factor. Although much progress has been made in automatically parallelizing compilers, they cannot handle the irregularities of typical desktop applications.

The process of manual parallelization typically involves identifying independent tasks within the program and assigning these tasks to threads. These tasks generally access many of the same data structures. These shared data structures must be manually identified and protected with locks to avoid data corruption. This process is difficult and error prone because it requires an understanding of the whole program. Furthermore, because each program consists of a different set of tasks, the process of parallelization must be repeated for each program parallelized. These two factors make developing multithreaded programs expensive, resulting in multithreaded programs being only developed for markets that are highly sensitive to performance (e.g., server workloads). In contrast, most desktop applications are purchased based on features, so most software vendors perceive the cost of parallelization to outweigh its benefits.

In contrast, we explore an alternate approach, called *Time-Shifted Modules* (TSM), that structures module interfaces to derive TLP from parallelism already present in programs. Large programs are frequently written as a series of *modules*, where each module performs a portion of the whole program's functionality. This approach improves testability and maintainability [18]; by providing all modules with well-defined interfaces and encapsulating their internal data, each can be independently tested and debugged, and modules can be modified in isolation without impacting the whole code base. Many (integer) programs perform a series of loosely-coupled computations at the granularity of hundreds or thousands of instructions, and frequently the boundaries between these computations correspond to module boundaries in the source code.

TSM is a programming technique for multithreaded processors that does not require the inclusion of additional hardware features. Module code is modified so that manipulations of encapsulated data can be performed safely in parallel with the main program. TSM uses 2 techniques (discussed in detail in Section 2) to expose TLP from an otherwise sequential program: (1) side-effect computations that are not needed immediately can be deferred using *time-shifting later*, and (2) function results can be computed early using *time-shifting earlier*. TSM provides the

appearance of a sequential execution by placing software synchronization at module interfaces to avoid race conditions. Module computations are scheduled on available threads in software (described in Section 3).

TSM has many benefits over traditional parallelization. TSM parallelization is simpler to perform because all of the inter-thread synchronization and communication occurs within a module; a programmer no longer must reason about the whole program. Furthermore, once parallelized, a module can be reused in other applications, amortizing the cost of parallelizing the module. In fact, because the interfaces of the parallelized modules maintain sequential semantics, a library of such modules could be created (much like the C++ standard template library (STL)) that could be used by programmers without having to reason about concurrency. The programmer creates what appears to be a sequential program, but the program executes faster when multiple thread contexts are available. To use such a library the programmer need not learn a new programming model nor a new programming language. For this reason, we think that TSM could be adopted on a large scale.

The two main drawback of this technique, with respect to traditional parallelization, are its limited scalability and applicability. It is not scalable because programs do not generally interact with enough modules to keep a large number of threads busy. This is not a major concern, however, because desktop machines — containing a single processor die with support for 2 to 4 threads — will only be able to exploit small degrees of parallelism. Applicability is a greater concern; for correctness this technique is restricted to modules that have limited interactions with other modules (discussed in Section 2.3). Despite this limitation, TSM can be applied to a number of important primitive data structures to significantly improve performance as evidenced by our case studies (Section 5).

This paper serves to demonstrate Time-Shifted Modules (TSM) and explore the performance benefits achievable using this technique. Furthermore, application of this technique results in workloads with fine-grain parallelism that push the envelope on synchronization and communication frequency. Because multithreaded processors and chip multiprocessors execute concurrent threads in close physical proximity, the latency of synchronization and communication can potentially be reduced to close to that of an L1 or L2 cache hit. This paper demonstrates that if architects provide high performance synchronization primitives, they can be used to achieve significant speedups.

The rest of the paper is organized as follows. In Section 2, we present a high level example of the parallelism we intend to exploit, and discuss some of the implementation details in Section 3. Using a simulated simultaneous multithreaded (SMT) processor (methodology is discussed in Section 4), we demonstrate the performance benefits of our technique in two case studies (Section 5). In Section 6, we discuss related work before concluding.

2 Time-Shifted Modules

Time-shifted modules enable concurrency by executing code from modules in parallel with code from the main thread. The technique exploits that modules frequently use data encapsulation; one of the module's functions must be used to access the internal data. This means that data structures encapsulated by the module can be in an inconsistent state until the module is accessed by the main thread. This allows module computations to be performed in the background (*i.e.*, *time shifted* with respect to the main thread), and by inserting synchronization into the interface functions we can be assured that the module is consistent before it is accessed by the main thread.

Module functions perform one or more of: (1) modify the module's state (*i.e.*, a side effect), and (2) return some data to be used outside of the module. These operations correspond to the two types of time shifts discussed in the next two sub-sections. In Section 2.3, we discuss the correctness requirements of this technique.

2.1 Time shifting later

In a sequential program, all of the side effects associated with a function are performed before the function returns, but this is not always necessary. It is not uncommon for functions to perform only side effects (when no value is returned) or for side effects to be independent of the return value (when an invariant is maintained or bookkeeping is performed). When this is the case, side effects need only be completed before their results are requested by the main thread.

In these instances, the side effects can be deferred, allowing the function to return earlier and the main thread to execute the rest of the program. The deferred side effects can be performed by a separate thread in the background. Because the module's data is encapsulated, synchronization can be employed on all functions that manipulate the module to ensure the side effect has been completed.

Figure 1. Time-shifting later delegates computing side-effects to a separate thread. (a) the original priority queue code executes the queue sort computation inline, (b) time-shifting later allows the main thread to execute block C while the sort is performed by another thread, (c) software synchronization is used to stall the main thread if it tries to remove an element before the sort is completed, but assuming sufficient buffering (d) the main thread never needs to stall on insertions.

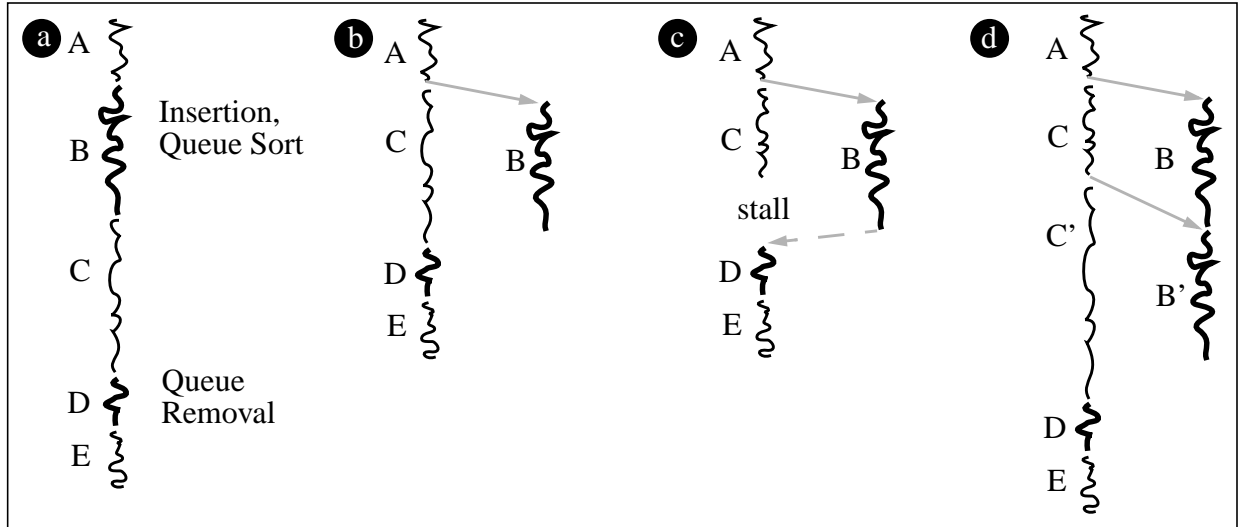
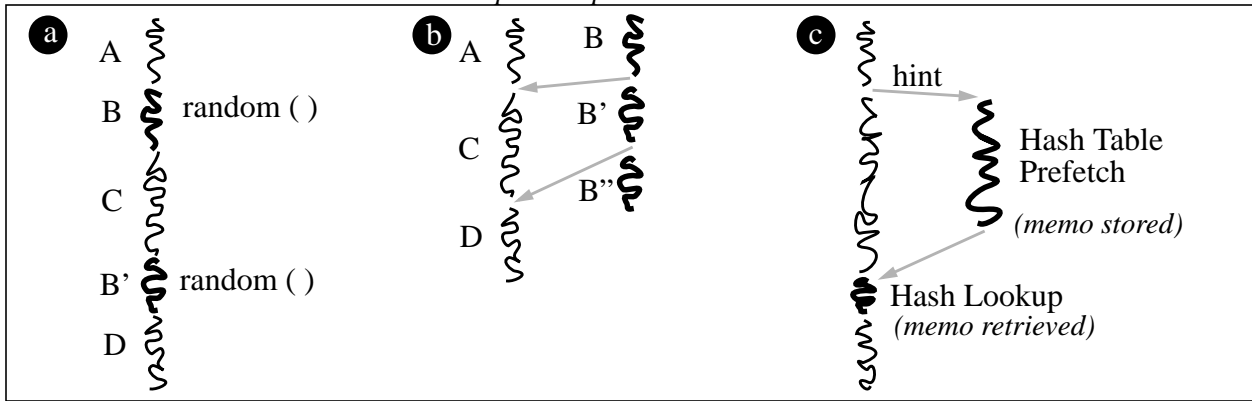


Figure 1 pictorially demonstrates how this technique can be employed on a priority queue module. A priority queue is a data structure that supports two main operations: insertion of an item with an arbitrary priority and removal of the highest priority item. Typically this data structure is implemented by keeping the queued items sorted — performing an insertion sort for each inserted item at the time of insertion — so that removals need only grab the item at the head of the sorted structure as is shown in Figure 1.a

As discussed above, the sort need not occur at the time of the insertion, but merely before the next removal. This allows us to delegate the sort to a second thread, performing it in the background, in parallel with other code executed by the main thread (Figure 1.b). Because we cannot be assured that the sort has completed by the time we want to remove the highest priority item, synchronization is used to indicate when it is safe for the main thread to perform the removal. If the priority queue is not ready when the main thread arrives, the main thread stalls on the lock (Figure 1.c). The main thread need never stall on insertions, assuming there is sufficient buffering to store the messages to the sorting thread (Figure 1.d).

This technique is commonly used in graphical user interfaces (*e.g.*, X Windows), where some display computations are delegated — through sockets —to the X server thread. It is also amenable to other output streams and to any module which requires maintenance computation, including

Figure 2. Time shifting earlier predicts that a computation will be required and pre-computes it: (a) stream producers, like a random number generator, have trivially predictable function call patterns, but sequential code must execute them inline, (b) by time shifting the code earlier, the random numbers can be generated in parallel with other code, (c) less predictable modules, like a hash table, will likely necessitate hints to enable results to be pre-computed.



sorted structures, memory allocators, red-black trees, etc. In these structures, the maintenance computation can potentially be over-lapped with unrelated computation.

2.2 Time shifting earlier

To complement deferring the computation associated with side effects, we'd like to reduce the time it takes to compute the results returned by module functions. This can be accomplished by executing the function before it is called by the main thread so that its results can be quickly retrieved when the function is called. Clearly this requires that the function called and its arguments are predictable.

One set of modules which are trivially predictable are those that generate a stream of data (i.e., modules that have one major method: "get next object"). Examples of such modules are random number generators, iterators, or any transformation applied to an input stream (e.g., tokenizers, uncompressors, decryptors, filters, etc.). Such modules can be speculatively iterated, enabling generating the stream to be overlapped with processing the stream. We use a software queue to decouple the producer and consumer, iterating the producer while the queue is not full.

Figure 2 demonstrates this technique for a random number generator. Generating high quality random numbers can be computationally intensive, but only the previously computed random number and a set of constants is required to compute each random number. In a sequential program, the computation of a random number must be performed inline (Figure 2.a), but clearly the stream of random numbers can be pre-computed and buffered. This pre-computation can be done

in a separate thread, in parallel with the main computation (Figure 2.b). When a random number is needed by the main thread it simply grabs the first one from the buffer.

Pre-computed results can be speculative in nature, in which case “architected” module state must be maintained and restored if a prediction was incorrect. In the random number generator example, if the random seed is updated, the pre-computed results are invalid. Because the current seed (i.e., the last number generated) is the module’s only state and this state is replaced by the new seed, no architected state needs to be maintained. The recovery consists of invalidating the pre-computed results.

The applicability of the time-shifting earlier technique can potentially be extended to more general modules by employing hints to instruct the module what result should be pre-computed. Such hints would indicate which function is likely to be called and what the parameters are likely to be. Hints can be inserted manually by the programmer, automatically by the compiler as a result of static analysis or profile-driven feedback, or by a dynamic/hardware mechanism like pre-execution [27].

Hints of this nature are frequently used to perform memory pre-fetching; constructing a module to execute speculatively enables the hints to be used for general pre-computation. For example, a lookup in a hash table typically requires key generation, a primary lookup into the hash table, and potentially a variable number of secondary lookups (either by walking buckets or by use of a secondary key). This process is impossible to prefetch using traditional means, but can be potentially performed in the background by a separate thread (Figure 2.c). The result could be stored in a buffer in the module, from where it could be collected by the main thread, much in the way that memoization can be used to avoid re-computation. The correctness of these predicted memoizations must be maintained (e.g., when items are removed from the hash table, corresponding memoized entries must be invalidated). Evaluation of such modules is left for future work.

2.3 Module Requirements

The computations performed by time-shifted modules do not occur when they would in a sequential program. To ensure that sequential semantics are maintained (i.e., the execution is equivalent to a sequential execution), we prevent multiple threads from accessing the same data structures simultaneously. This prevents a time-shifted computation from accessing another module, unless that module is completely encapsulated within the time-shifted module (and hence

cannot be accessed asynchronously by the main thread) or the module is independently synchronized and its behavior does not depend on access order (like a memory allocator). Note that benefit can be achieved by time-shifting a subset of a module's functions, when some functions preclude the optimization of the whole module. This restriction does limit the technique, but we have found it to be applicable to a number of important primitive data structures where programs spend significant amounts of time.

3 Implementation Details

The grain size of parallelism that can be exploited with our technique depends partly on the overhead of synchronization and scheduling. In this section we describe the structure of the parallelized modules and explore these overheads. Note that our implementation is intended for a SMT processor and assumes that the L1 cache is shared by the cooperating threads and hence does not perform prefetches to tolerate inter-cache transfer latencies. An implementation that efficiently supports chip multiprocessors is an area of future research. The overhead has 3 main components: (1) acquiring/releasing locks, (2) fine-grain scheduling of available computations on threads, and (3) evaluating state of the module. We discuss these overheads in this section and quantify them in our case studies.

Generally, the parallelization of a module extends its data footprint to include a communication region. This communication region is used by the main thread to deposit "commands" for the module or collect pre-computed results from it. The communication region and the rest of the module are protected by separate locks to enable the main thread to communicate with the module while it is being manipulated by another thread. A typical lock acquire/release sequence is shown in Figure 3.a. The instruction overhead of this sequence is 6 instructions if the acquire succeeds.

Because there can be more modules than there are threads available, it is necessary to schedule the modules on to the available threads. We do this using a centralized work list. When the main thread desires to delegate work, it sets a "dirty bit" in the module and posts the module to the work list. Available threads monitor the work queue, and when work is posted, remove an assignment from the queue. When the work is completed, the dirty bit is cleared. The dirty bit is used to prevent the same module from being posted to the work list multiple times. The code sequences to post/retrieve a module to/from the work list are shown in Figure 3.c. The post and retrieve code sequence are 11 and 30 dynamic instructions, respectively.

Figure 3. Synchronization and scheduling instruction overhead: (a) code sequences for successfully acquiring and releasing lock, (b) global structure definition for requests for background processing (our implementation reserves the zero offset of a time-shifted module for the helper thread function, allowing a single address to specify both the module's address and the function to be called), (c) C code for main thread inserting a **request** into the queue, and the main helper thread event loop, where helper threads arbitrate for entries in the request queue, wait for requests, and process the request.

<pre> a acquire: ldl_l t0, 0(a0) blbs t0, failed bis t0, 0x1, t0 stl_c t0, 0(a0) beq t0, failed release: stl zero, 0(a0) </pre>	<pre> c request: if (rq->head < (rq->tail + QUEUE_SIZE-3)) { rq->data[head & QUEUE_MASK] = object; rq->head ++; } </pre>
<pre> b typedef void (*func) (void *); struct request_queue_t { func *data[QUEUE_SIZE]; int head; /* producer */ int tail; /* consumer */ int lock; }; struct example_TSM_t { func helper_function; ... }; </pre>	<pre> helper thread event loop: while (1) { get_lock_Q(rq->lock); int my_entry = rq->tail; rq->tail = my_entry + 1; release_lock(rq->lock); volatile func **d = (volatile func **) &rq->data[my_entry & QUEUE_MASK]; QuiesceWhileEqualQ(*d, 0); func *data = *d; func f = *data; *d = (func *)0; /* clear entry */ f ((void *)data); } </pre>

When the main thread has delegated work to another thread and desires to access the module again (in a manner that requires the delegated work to be completed) it waits for the other thread to complete its work. Because the delegated work may not yet have been begun by another thread (either because the other threads are busy with other work, or because no other threads are currently available) the main thread first checks to see if the work is in progress (by trying the lock on the module proper). If the work is not in progress, the main thread first performs the deferred work and then manipulates the object in originally desired manner.

When a computation is delegated to another thread, that thread has to collect information from the command region and determine what manipulation needs to be performed on the module. The overhead incurred in evaluating the state of the module will differ for each parallelized module.

4 Methodology

We perform our performance analysis on a simulated SMT processor configured to approximate our expectations for the upcoming Alpha 21464, extrapolating from the design of the 21264 [20] and what has been announced about the 21464 [14]. Details can be found in Table 1. The rest of this section discusses the quiesce instruction, efficient synchronization, and issues relating to parallelization on SMT and CMPs.

Quiesce. Our implementation includes the *quiesce* instruction, as described in [14]. This instruction prevents threads from consuming execution resources while waiting for a synchronization event (*i.e.*, spinning on locks, semaphores, and barriers). The quiescing thread specifies the memory address it is waiting on, and is put to sleep until that memory location is modified. All of our synchronization primitives quiesce if the desired lock is not available.

Efficient Synchronization. The main deviation our simulation model makes from our expectations of the 21464 is in support for synchronization. The Alpha supports a relaxed memory ordering model that requires software barriers to indicate when ordering is required. Historically, because threads synchronized infrequently, barriers could be implemented as costly operations, stalling the pipeline for many cycles [20].

Table 1. *Simulation parameters approximating expectations for the Compaq Alpha 21464.*

Front End	A 64KB 2-way set associative instruction cache, a 64Kb YAGS [13] branch predictor, a 32Kb cascading indirect branch predictor [12], and a 64-entry return address stack. The front end can fetch past taken branches. A perfect BTB is assumed for providing target addresses, which are available at decode, for direct branches. All nops are removed without consuming fetch bandwidth.
Execution Core	8-wide (fetch, decode, execute, retire) machine with a 128-entry instruction window, a full complement of simple integer units, 4 load/store ports, and 3 complex integer units, all fully pipelined. The pipeline depth (and hence the branch misprediction penalty) is 14 stages. For simulation simplicity, scheduling is performed in the same cycle in which an instruction is executed. This is equivalent to having a perfect cache hit/miss predictor for loads, allowing the scheduler to avoid scheduling operations dependent on loads that miss in the cache.
Caches	The first-level data cache is a 2-way set-associative 64KB cache with 64 byte lines and a 3-cycle access latency, including address generation. The L2 cache is a 4-way set-associative 2MB unified cache with 128-byte lines and a 6-cycle access. All caches are write-back and write-allocate. All data request bandwidth is modeled, although writeback bandwidth is not. Minimum memory latency is 100 cycles.
Prefetch	In parallel with cache accesses, a 64-entry unified prefetch/victim buffer is checked on all accesses. A hardware stream prefetcher detects cache misses with unit stride (positive and negative) and launches prefetches. In addition, when bandwidth is available, sequential blocks are prefetched (before a stride is detected) to exploit spatial locality beyond 64 bytes.

In order to benefit from fine-grain parallelism, the cost of synchronization must be low. Making synchronization cheap has two requirements: maintaining a specified memory ordering must be cheap, and synchronization primitives (e.g., load-locked, and store-conditional) must be implemented efficiently. Existing processors with stronger memory ordering models [33, 19] have already demonstrated that memory ordering can be implemented efficiently. We believe that there is no fundamental problem with efficient implementation of synchronization primitives (assuming they have been correctly architected).

Our implementation of synchronization primitives is aggressive and is natural extension of existing load/store queues. Our processor implements Total Store Ordering (TSO) [32] as its consistency model. Retired stores are written into a write buffer from which they are made architecturally visible in program order. On a coherency event (including a store leaving another thread's write buffer) all in-flight loads (that have received their value) are snooped. If one has received a stale value, the instruction is replayed by squashing the thread at that point.

Parallelization and SMT. Because all threads in an SMT processor share the same pool of execution resources, parallelization should not be performed naively. If a program (e.g., a regular, computation-bound scientific program) can effectively utilize most of the processor's resources, the overhead of synchronization and communication might make a parallelized version of the program slower. For the irregular, integer programs we are considering, this is less of a concern.

SMT vs. CMP. The benefit of parallelization for SMT differs widely from the benefit for a CMP. In an SMT processor, a single thread can use all of the available resources, but may not use them efficiently. A single thread cannot tolerate control-flow mispredictions or instruction cache misses, and the finite instruction window limits the amount of memory latency it can tolerate. When multiple threads are running, the processor need not speculate as much, reducing the cost of misspeculations (like branch mispredictions), and other threads can continue to make progress when one thread is stalled on an instruction or data cache miss. In addition, multiple threads might have more independent operations from which to schedule and better tolerate engineering design constraints (e.g., clustered register file).

On a CMP, parallelization provides the program access to additional execution resources. Although these additional resources creates the potential for a larger speed-up, the cost of communicating and synchronization is likely to be higher than for an SMT. Typically, CMP's don't

share their L1 data caches, necessitating the cache coherence mechanism to be involved in all inter-thread interactions.

5 Case Studies

In this section, we discuss two instances where we used TSM: an example of each of time-shifting later and earlier. In Section 5.1, we discuss parallelizing the heap data structure (a priority queue) in the SPEC2000 integer benchmark `vpr`. Section 5.2 presents the parallelization of `libexo` (a library for packing/unpacking data structures to/from persistent storage) used by the pervasive micro-architectural simulator SimpleScalar [6].

In both cases, we compare our modified code to the original code, so we attempted to avoid changes to the code (other than those required for parallelization) that might affect its performance. Each case study took on the order of a week to analyze the code, parallelize it, and perform rudimentary tuning by a programmer with limited experience writing multithreaded applications.

5.1 Priority queues (VPR)

The SPEC2000 integer benchmark `vpr` is a CAD tool for automatically placing and routing electrical circuits on field-programmable gate arrays (FPGA). In the routing phase, the set of currently reachable wire segments are tracked along with an associated “cost” value which estimates the congestion of the wire segment [3]. A priority queue is used to select wire segments in order to find a route which minimizes congestion. The priority queue’s order is maintained using a heap *sort* [8]. A heap requires a “heapify” operation to be performed after both insertions and removals to maintain the heap invariant. In `vpr`, the queue is frequently accessed and often contains hundreds or thousands of elements, causing this heapify operation to be responsible for approximately half of the instructions executed by the program. The average heapify operation takes over a hundred instructions and frequently causes branch mispredictions and cache misses.

5.1.1 Implementation

As was demonstrated in the example in Section 2.1, this sort can be performed in the background. For this case study, we refactored the heap code to include a pair of queues to decouple sorting the heap from the main thread. We undertook two implementations. The first, which we’ll call *simple*, was a minor modification of the original code which defers heapify operations, but

requires them to be complete before the next removal from the heap. The second implementation, which we'll call *enhanced*, increases the concurrency by exploiting the fact that only the head of the priority queue must be known in order to perform a removal.

Both implementations include an insertion queue that holds pending insertions. The main thread can asynchronously place values to be inserted in this queue, while the background thread performs the insertions and their associated heapify's.

In the simple implementation, before a remove is performed all pending heapify's must be completed. The remove takes the element from the head of the heap, deferring the associated heapify. The enhanced implementation adds a second queue to hold the head (up to 8 elements) of the priority queue in sorted order. By ensuring that the head queue is always valid, the main thread can remove an item while insertions are still pending. Ensuring validity requires comparing inserted elements to the tail of the head queue, and performing an insertion sort into the head queue if the element's priority necessitates it.

5.1.2 Results

We evaluated our modified version of *vpr* using the ref input parameters modified to reduce execution time (*net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 25 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 100 -initial_pres_fac 1000*). Our simulations skip over the first 575 million instructions (where the two programs are exactly the same) and then run to completion (about 7 billion instructions in the base case).

Despite the fact that the parallelized binary executes more instructions, its multithreaded nature provides latency tolerance, enabling 8% and 26% speedups¹ for the simple and enhanced cases respectively. As can be seen in Table 2, the parallelized versions are able to achieve aggregate IPCs 31% and 63% greater than the sequential version, outweighing the 20% and 26% additional instructions these versions require. All versions have pretty equivalent numbers of L1 cache misses, but the parallelized versions have significantly more branch mispredictions. Many of these additional mispredictions are due to inter-thread communication, either from synchronization

1. Additional instructions incurred from spinning are included in both IPCs and retired instructions, but because all of our synchronization primitives use quiesce, very little "spinning" takes place. Running the benchmarks to completion allows execution performance to be meaningfully compared.

primitives themselves or the additional unpredictability induced by asynchronous threads modifying the same data structure.

Table 2. Statistics for original, simple parallel, and enhanced parallel versions of vpr. The parallel cases have some statistics broken down by thread using the notation [main thread/helper thread].

	original	simple	enhanced
cycles (billions)	4.3	4.0	3.4
instructions retired (billions)	7.0	8.4 [5.2 / 3.2]	8.8 [4.4 / 4.4]
total instructions fetched (billions)	18.7	20.0	18.8
IPC (instructions/cycle)	1.6	2.1 [1.3 / 0.8]	2.6 [1.3 / 1.3]
branch mispredictions (millions)	47	65 [31 / 34]	61 [32 / 29]
L1 misses (millions)	131	130	147
synchronization overhead (millions of instructions.)	0	295	408
scheduler overhead (millions of instructions.)	0	410 [164 / 246]	86 [65 / 21]

We have attempted to estimate the synchronization and scheduler overheads observed by the parallelized versions, by measuring the dynamic frequency of static instructions involved in synchronization and scheduling. Since these instruction categorizations were done by hand, these results are approximate. In both cases, the combined synchronization/scheduling instruction overhead is less than 10%. Because the enhanced case is more decoupled than the simple case, its helper thread can keep itself busy managing the heap for long stretches of execution, requiring less scheduling but increasing the contention for locks. The remaining discrepancy in instructions counts can be attributed to overhead necessary to evaluate the module’s current state and for additional address generation and memory operations due to inter-thread communication being performed through memory.

5.2 File output (Libexo)

Libexo is library which enables arbitrary tree-like data structures to be stored persistently (i.e., on disk as *exo files*) and loaded back into memory. Data is stored in ASCII, enabling the data structures to be passed between architectures. Libexo is used in the simple scalar microarchitecture simulator to provide the External I/O (EIO) functionality to store checkpoints and log syscall interactions. EIO enables repeatable simulations, running on platforms that cannot emulate Unix system calls, starting from checkpoints, and sampling.

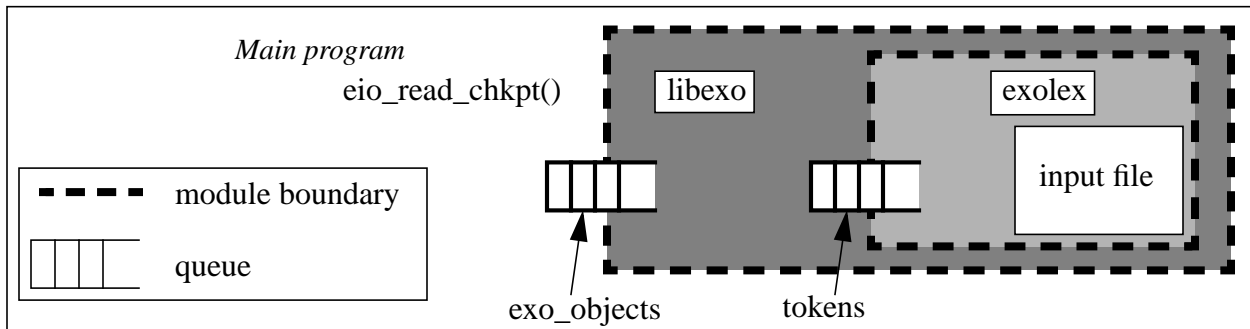
5.2.1 Implementation.

Libexo consists of two parts: routines for reading *exo* files and those for writing them. For this case study, we parallelized the routines for reading *exo* files because this is the more common operation; the routines for writing *exo* files are also parallelizable. Reading an *exo* file consists of two main steps (shown in Figure 4): 1) the lexical analysis of the file's contents, and 2) the construction of *exo objects*. In the original code, these steps were performed by different modules; the lexical analyzer was automatically generated using flex [1], and the *exo* object constructor consisted of about 10 hand-written functions. In our parallelized version of the code, we maintain these module boundaries, but modify the interfaces slightly.

Both modules produce streams of data: the lexical analyzer (`exollex`) produces a stream of tokens and their associated text, and the object constructor produces a stream of *exo* objects. On average, it takes `exollex` roughly 70 instructions to generate a token, and `libexo` close to a million instructions to generate an *exo* object. The average *exo* object consists of thousands of tokens.

We modified the modules to allow their streams to be produced and consumed concurrently, by using queues to buffer the communication. The resulting execution is a pipeline of 3 threads: the `exollex` module produces tokens, the `libexo` module converts these tokens to *exo* objects, and the *exo* objects are used by the main thread. This transformation was very straightforward for the object construction routines, but more complicated for the lexical analyzer. The code which flex generates uses globals extensively and prevents concurrency by temporarily modifying the input text buffer (by null terminating the token's string) and exposing it as part of the interface. We rewrote the code to encapsulate the global data and copy the token text to a separate buffer.

Figure 4. Parallelized code contains two nested time-shifted modules. An input file is lexically analyzed by the `exollex` module and the resulting tokens are exported through a queue. The `libexo` module uses these tokens to construct *exo* objects, which it exports through a queue to external I/O (`eio`) functions. Because the `exollex` module is completely encapsulated in `libexo`, `libexo` is free to access it speculatively.



Unlike a true stream producer, the lexical analyzer’s interface includes more than a simple “get next” function. The other functions query the token stream, and were rewritten to access pre-computed tokens in the buffer.

5.2.2 Results

We evaluated our modified version of `libexo` using `sim-eio`, a functional simulator. Because we only modified code related to loading the checkpoint, we configured our simulations so that this portion of the code dominates the execution (`Sim-eio` is instructed to execute only 1000 instructions after loading the checkpoint for the SPEC2000 benchmark `gcc`). This allows us to simulate the benchmark to completion. Clearly, this is only one phase of the program’s execution; the other phases would have to be parallelized independantly.

Again, the parallelized version is able to achieve a much higher aggregate IPC (up 65%) at the cost of increasing the instruction count (up 19%), resulting in a 39% speedup (as shown in Table 3). The `libexo` module performs most of the computation, with the main and `exolex` (the lexical analyzer) threads periodically quiescing when buffers are empty and full respectively. Unlike `vxr`, because the communication is much more regular (each thread is simply reading a result out of a queue), there is not a substantial increase in the number of branch mispredictions. This, coupled with SMT’s ability to tolerate branch mispredictions (because it speculates less), allows the parallelized version of `libexo` to fetch significantly fewer instructions than the non-parallelized version. Because the `libexo` thread is kept busy (i.e., it always has free buffers into which it can write) it never needs to reschedule. Periodically, the `exolex` module fills its buffer, quiesces, and then must be rescheduled by the `libexo` thread when the token buffer is empty. This happens rarely enough that the scheduling overhead is small (less than 0.1 percent).

Table 3. Statistics for original, simple parallel, and enhanced parallel versions of `sim-eio`. The parallel cases have some statistics broken down by thread using the notation [main / exolex / libexo].

	original	parallel
cycles (millions)	228	164
instructions retired (millions)	843	1000 [93 / 367 / 540]
total instructions fetched (millions)	1,812	1,309
IPC (instructions/cycle)	3.7	6.1 [0.6 / 2.2 / 3.3]
branch mispredictions (millions)	1.8	1.9 [0.0 / 0.0 / 1.9]
L1 misses (millions)	0	0
synchronization overhead (millions of instructions.)	0	26.3 [0.0 / 11.0 / 15.3]
scheduler overhead (millions of instructions.)	0	0.85 [0.23 / 0.62 / 0.0]

The synchronization overhead comes almost exclusively from reading and writing the token queue, as very few *exo* objects are produced.

6 Related Work

The concept of module-based parallelization has previously been explored in other contexts. For the most part, these other contexts can exploit more general classes of parallelism, but require the programmer to change their programming model. Hardware schemes to exploit this parallelism (through speculative parallelization) are similarly more widely applicable, but require non-trivial hardware mechanisms to verify the correctness of the speculation.

Probably the closest related work is the concept of a *future* [16]. Futures allow *closures*, procedures with arguments, to be handed to a runtime system, which can schedule them for parallel execution if execution resources are available. When the original thread needs the value of a future, it either waits until the value is computed or computes the value itself if nobody else has started working on the task. Futures were originally proposed in the context of functional languages (MultiLisp), but can be implemented in a message passing fashion in imperative languages like C and Fortran with Linda [7]. A related concept are the continuations implemented in Cilk [4], which allow incomplete closures to be created; when a closure is later completed it becomes available for scheduling by the runtime.

In the era of massively parallel computers, there was a lot of work done in the PL community exploring how languages should support parallelism. One proposal, Actors, advocates that objects should be independent entities that communicate with each other asynchronously through messages [2]. In such a system, the computations performed by different actors can be executed in parallel.

There has been extensive work in functional languages to automatically extract task-level parallelism [17]. These techniques can likely exploit the same parallelism exploited by TSM for programs written in functional languages, but are not applicable to imperative languages like C/C++/Java.

Although these approaches are scalable ways of expressing parallel computations, each require learning a different programming model. In contrast, TSM libraries can be used by programmers

only familiar with a sequential programming mindset. This accessibility comes at the cost of limited scalability and applicability.

Another approach to deferring side-effect computation is proposed in [25] where the reference counting computation for a garbage collector is performed in a parallel thread. This parallel thread executes a reduced version of the original program (created through program slicing) that periodically reads results logged by the main thread.

The decoupled access-execute architecture [28] decomposes a single sequential program into two threads: one responsible for generating memory addresses (the access stream) and one manipulating the loaded values (the execute stream). In programs where address generation is largely decoupled from other computation, these two streams could slip with respect to each other, enabling memory latency to be tolerated. TSM also decouples loosely coupled computations, but does so at a larger granularity and is not limited to address computations.

Multiscalar [29] (and other speculative multithreading proposals) exploits the parallelism in sequential programs by speculatively parallelizing them. The hardware must buffer all speculatively modified memory state and detect any memory ordering violations. Time-shifted modules avoid this additional hardware at the expense of some additional programmer effort; they exploit the encapsulation of data (and hence are only applicable when such encapsulation exists) and explicit synchronization to ensure memory violations will not occur. A previous proposal for software-only speculative parallelization exploits stride-predictable memory access patterns in scientific codes to make detection of memory ordering violations feasible [5].

Speculative Data-driven Multithreading (DDMT) [27] is an enhancement to an SMT core that allows portions of a program to be speculatively computed early (in data-driven threads) and if the speculation was correct the results of the computation can be used by the main thread. Although this technique avoids the main thread from having to execute the computation, the computation must still be fetched by the main thread (in order to verify that the speculation was correct) which limits the speedup achievable. Time-shifted modules have instruction overhead in the form of explicit synchronization, but if a computation is correctly speculated it is only fetched and executed once. The fixed overhead of TSM make it most applicable for larger computations, whereas data-driven threads (which need not be contiguous) are more suited for smaller, high-impact code fragments like slices of cache missing loads and mispredicting branches.

Previous studies exploring the performance of multithreaded code with SMT include: databases [22], web serving [26], SPLASH [23], and fine-grain scientific programs [31]. To our knowledge this paper includes the first analysis of fine-grain integer programs on an SMT.

7 Conclusion

In this paper, we have explored a technique to exploit the thread-level concurrency that will be available in next generation processors. This technique exploits the modularity of programs and data encapsulation to enable portions of otherwise sequential programs to be delegated to other threads. Our technique has two main advantages over traditional parallelization techniques: (1) it is simpler to perform because only code within the parallelized module need be considered, and (2) once parallelized a module can be reused in other programs. These advantages come at a cost of reduced scalability and the need for efficient synchronization. These characteristics make the technique a good fit for multithreaded processors and chip multiprocessors, which execute a small number of threads in close physical proximity.

We demonstrated the technique through two case studies, one for each class of “time shift.” The priority queue example demonstrated how side-effects can be deferred using the “time shift later” technique. The libexo (file parsing) example showed how results can be pre-computed using the “time shift earlier” technique. Our simulations of a simultaneous multithreading (SMT) processor indicate that speedups of 26% and 39% can be achieved through the use of this technique, without any additional hardware. In fact, we may be under-estimating the performance benefit; our simulations do not model many engineering details of the processor that potentially prevent a single thread from achieving high performance [9], but that can be tolerated through multithreading.

Although not all programs will benefit from this technique (nor perhaps even all phases of the programs that do benefit), we believe that it is a source of parallelism that programmers and architects alike should consider. As the number of transistors on a chip continues to increase, micro-processor designers will continue to incorporate techniques like multithreading and chip multiprocessing in there designs. In as little as five years time, it may be difficult to buy a true uni-processor. The current state of software does not effectively exploit concurrency. Time-shifted modules is an evolutionary technique that can potentially help bridge the gap.

8 References

- [1] Flex: a fast lexical analyzer generator. <http://www.gnu.org/software/flex/flex.html>.
- [2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [3] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *Seventh International Workshop on Field-Programmable Logic and Applications*, pages 213 – 222, Sept. 1997.
- [4] R. D. Blumofe, C. F. Joerg, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pages 207–216, July. 1995.
- [5] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based Speculative Parallelism. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.
- [6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.
- [7] N. J. Carriero and D. H. Gelernter. Linda in Context. *Communications of the ACM*, 32(4), Apr. 1989.
- [8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [9] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.
- [10] K. Diefendorff. Compaq chooses SMT for Alpha. *Microprocessor Report*, 13(16), Dec. 1999.
- [11] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), Oct. 1999.
- [12] K. Driesen and U. Hoelzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proc. 31st International Symposium on Microarchitecture*, pages 249–258, Dec. 1998.
- [13] A. Eden and T. Mudge. The YAGS Branch Prediction Scheme. In *Proc. 31st International Symposium on Microarchitecture*, pages 69–77, Nov. 1998.
- [14] J. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, Oct. 1999.
- [15] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread Level Parallelism and Interactive Performance of Desktop Applications. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–138, Nov. 2000.
- [16] R. H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Aug. 1984.
- [17] K. Hammond. Parallel Functional Programming: An Introduction. In *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, Sept. 1994.
- [18] A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000.
- [19] K. Hurd et al. A 600MHz PA-RISC Microprocessor. *International Solid-State Circuits Conference*, Jan. 2000.
- [20] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.
- [21] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Characteristics of Desktop Applications on Windows NT. In *Proc. 25th International Symposium on Computer Architecture*, Jun. 1998.
- [22] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proc. 25th International Symposium on Computer Architecture*, pages 39–50, Jun. 1998.
- [23] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting Thread-level Parallelism into Instruction-level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computers*, 15(2), Aug. 1997.
- [24] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

- [25] M. Plakal and C. N. Fischer. Concurrent Garbage Collection Using Program Slices on Multithreaded Processors. In *The International Symposium on Memory Management (ISMM)*, Oct. 2000.
- [26] J. A. Redstone, S. J. Eggers, and H. M. Levy. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–256, Nov. 2000.
- [27] A. Roth and G. Sohi. Speculative Data-Driven Multi-Threading. In *Proc. 7th International Symposium on High Performance Computer Architecture*, Jan. 2001.
- [28] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th International Symposium on Computer Architecture*, Jul. 1982.
- [29] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.
- [30] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.
- [31] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proc. 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.
- [32] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.
- [33] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.